Pioneer

Mobile Robots

with Pioneer Server Operating System Software

PAI Programmer's Manual

Contents

Page

1. Introduction to PAI

PAI – the Pioneer Application Interface – is a library of C functions and definitions for control of Pioneer Mobile Robots. Written by Barry Werger of the University of Southern California, PAI works in conjunction with SRI International's Saphira robotics application-development environment, collecting its many low-level robot control features into simpler and more easily managed APIs. PAI is not intended to replace Saphira, but to smooth the learning curve for operation of Pioneer Mobile Robots, as well as development of Saphira-enabled robot applications.

1.1 Where to get PAI

PAI comes with your Pioneer robot. Updates and new versions of PAI are available *free* to Pioneer owners from our Internet support website. Please be sure to use the special `username & password` that accompanied your Pioneer robot or Saphira software to gain access to the restricted files:

`http://robots.activmedia.com`

There are two PAI files: **`PAI.tgz`** is the GZIP'd and tar'd archive for Unix/Linux users; **`PAI.zip`** is for Microsoft Windows 95/NT users.

1.2 Related Resources

1.2.1 Manuals

PAI depends on a working knowledge of the Pioneer Mobile Robot—how to turn it on, enable the motors, connect the radio modems, and so on. Consult your ***Pioneer Operations Manual*** for details.

You should also have a copy of the ***Saphira Software Manual*** on hand, too. It provides an excellent overview as well as details on operation of the Saphira client software.

If you have a Gripper and/or Experimenter's Module, consults this Pioneer accessory's Manual for port assignments.

Obtain copies of the latest Pioneer and Saphira Manuals (this document, too) from our Internet site:

`http://robots.activmedia.com`

Fast-Track Vision System (FTVS) owners should have a copy of the User's Guide handy. Newton Labs, developer of the Cognachrome Vision System which is the native vision platform behind the FTVS also provides extensive documentation at their website:

`http://www.newtonlabs.com`

1.2.2 Development Platforms

Linux- and Unix-based PAI and Saphira programs get compiled with either cc or gcc. Programs that use the included GUI interface also require the X-Window system and the `libXm.so` and/or `libXm.a` Motif libraries.

Microsoft Windows95/NT developers need access to Microsoft Visual C++ version 4.x or later; we don't support Borland C/C++ .

1.2.3 Newgroups

We announce PAI and Saphira updates and new versions, as well as share ideas and code, through two main email-based newsgroups: `pioneer-users@activmedia.com` and `saphira-users@activmedia.com`. To join—and please do join—send an email message to `pioneer (or saphira)-users-request@activmedia.com` with the Subject: **subscribe**. The newsgoup manager will reply with sign-up instructions and lots of other helpful tips.

1.2.4 Support

If something seems (or clearly is) broken with PAI, Saphira, or your Pioneer robot, send an email message to their appropriate support group and a team of experts will leap to the rescue:

`pioneer-support@activmedia.com`

-or-

`saphira-support@activmedia.com`

2. Programming with PAI

2.1 Installing PAI

You must have Saphira version 5.3 or later for PAI to work. (We recommend later Saphira versions 6.x.) If you do not, obtain a copy of Saphira for your preferred operating system from the Pioneer Support Website. Don't forget to use the special **`username & password`** that accompanied your Pioneer robot or Saphira software to gain access to the restricted files:

`http://robots.activmedia.com`

Always install Saphira first, then add the PAI archive into the top-level directory of Saphira – commonly **`/usr/local/saphira/ver61`**.

2.1.1 Windows95/NT

Double-click the `pai.exe` file to activate the auto-extractor. Place the files into the top-level directory of your Saphira distribution. For convenience, you might also copy the dynamic-link load file `pai\object\pai.dll` into the `windows\system` directory of your boot drive.

2.1.2 Unix

```
% gunzip pai.tgz
% tar -xvf pai.tar
```

The "%", of course, is the shell command prompt. Don't type it.

2.1.3 Linux

```
% tar -zxvf pai.tgz
```

2.2 PAI Directory Structure

Once installed, the fresh PAI directory should contain these following subdirectories and files:

```
saphira/ver61/(lots of other directories and files)
             /pai/examples/ulogo.c (fun partial implementation)
                          /makefile (or ulogo.mak)
                 /include/pai.h (include this with your programs)
                 /object/pai.o (or pai.dll)
                        /pai.rc (Win32 version)
                 /source/pai.c (yup, it's the pai source file)
                /development/(your own source/object/executable files)
                           /makefile (or .mak file for Win32)
```

2.3 Creating PAI Programs

To use PAI, simply include the PAI header file in your C-language source file.

2.3.1 PAI Source and Header

```
#include "pai.h"
```

Compose your programs using the "pai" functions and variables, as well as Saphira-based "sf " ones, according to the PAI reference guide that follows and the information found in the *Saphira Software Manual*. Include the `pai.h` header file in client programs. And, of course, feel free to examine and modify it and the source program, `pai.c`, that comprise the core of PAI.

2.3.2 Environment Variables

Your compiler, Saphira, and PAI systems need to know where you put your files. And the applications depend on various dynamically link-loaded libraries. To account for these, you must point the **SAPHIRA** and, with Linux/Unix systems, the **LD_LIBRARY_PATH** environment variables to PAI and other Saphira directories and files.

With Unix/Linux systems, use **setenv** or **export**. For example:

```
% setenv SAPHIRA /usr/local/saphira/ver61/  (the final slash is important here)
% setenv LD_LIBRARY_PATH=$(SAPHIRA)/handler/obj/
                -or-
% export SAPHIRA=/usr/local/saphira/ver61
% export LD_LIBRARY_PATH=$SAPHIRA/handler/obj
```

With Windows95/NT systems, add SET SAPHIRA=*path* to your **C:\AUTOEXEC.BAT** file, replacing *path* with the actual pathname to Saphira's top-level directory on you system. For example,

```
SET SAPHIRA=C:\saphira\ver61
```

Also, make sure copies of the pai.dll and sf.dll are in windows/system or in the root directory of your executable. Unix and Linux users, you may want to modify your makefile to automatically set the Saphira path. See the very next section.

2.3.3 Unix/Linux PAI Applications

We've included a standard `makefile` in several places for Unix/Linux users to take the pain out of compiling and linking PAI-based applications. They are **`examples/makefile`**, **`source/makefile`** and **`development/makefile`**. Make copies for yourself and place them in your own PAI development directories. The makefile is portable – copies may exist nearly anywhere in your filesystems because it references the necessary libaries and header files through the SAPHIRA envionment variable you set as per section 1.5.2.[1]

The handy makefile is for individual PAI-based programs, but feel free to use it as the basis for construction of a makefile for more complicated programs. It will produce and executable from any PAI-based source file that does not require extra libraries, as long as a copy of this makefile resides in the same directory as the file to be compiled. To compile **`filename.c`**, for example, use the command:

```
% make filename  (filename without the extension!)
```

-or-

```
% gmake filename (if make returns errors on your system)
```

If you don't have access to gmake, have a talk with your system administrator.

2.3.4 Makefile (Windows95/NT)

We've included two different makefiles (".mak") for Windows95/NT developers in the `\examples` directory of PAI. Use these as the bases for your own Visual C++ projects. The first, `ulogo.mak`, contains the components you need to create a PAI-based "Console Application" – in this case, one called `ulogo.exe`.

The other makefile, `source/`**`pai.mak`**, creates the **`pai.lib`** and **`pai.dll`** files you'll need for your applications.

2.3.5 Window and Non-Window Modes

PAI through Saphira provides a window-based Graphical User Interface for your applications, which lets users see the robot in action, its various status values, sonar artifacts, and so forth, as well as manipulate various settings, including the connection channel. We discuss the differences and utilities between window ("GUI Application") and non-window ("Console Application") modes in detail in the *Saphira Software Manual*.

Suffice it to say that you may include the Saphira control window in your applications at compile/link time by adding the **`MODE=WINDOW`** argument to the `make` or `gmake` command (Linux/Unix):

```
% make filename MODE=WINDOW
```

The default is MODE=NOWINDOW.

For Microsoft Visual C++ developers, copy the appropriate "Application" or "Console Application" project for window versus non-window PAI-based applications. Notice that the window application project includes the resource file, `include\pai.rc`.

[1] Add the export or setenv command line to the makefile so that it will automatically set the SAPHIRA variable for you.

2.4 Sample PAI Application

The `examples` directory contains a sample program called **`ulogo.c`** which gives you a good idea of the simplicity and use of PAI. Now is a good time to dust off your computer and compile/link the program as per the instructions in section 1.5 to test your system. Also, please read the **`ulogo.txt`** file in the same examples directory to find out how to operate the resulting ulogo (.exe) program.

2.5 Possible Problems

The source of most problems with Saphira/PAI is with the environment variables. Missing or erroneous references have sometimes bizzare and usually misleading effects. Please make it the first thing you check when experiencing problems with Saphira/PAI, for instance:

```
% echo $SAPHIRA
```

to see what SAPHIRA is set to on your system.

Another leading problem is not understanding which port to connect the PAI/Saphira client with the Pioneer or another robot server like the simulator. Start by connecting with the simulator. It's the most direct connection and can help you eliminate system setup-related errors. The Linux/Unix simulator opens a communications file called `robot` in `/tmp`, so make sure you have access to `/tmp` and that the `robot` file isn't already in use by someone else.

Then, once you are sure your PAI program works with the simulator, connect the client with your Pioneer Mobile Robot, typically through the serial port COM1 or COM2 (/dev/ttyS0 or /dev/ttyS1, or /dev/cua0 or /dev/cua1, respectively). And make sure that you have access privileges to the serial port. Often, Unix/Linux users do not have general access privileges to the serial ports (see `paiRobotStartup` below).

3. The PAI Standard Library

Many of the constants and variables encountered by PAI programmers are defined in the various $**SAPHIRA/handler/include/** header files. We will point out the options and values you have when using the PAI functions, but it wouldn't hurt to have a freshly printed copy of the *Saphira Software Manual* handy, too.

Also, notice that PAI functions and variable names begin with the prefix "pai", whereas Saphira-related things start with "sf".

3.1 Client-Server Connection

For non-window-based ("console") PAI applications, the first act your main program typically must accomplish is make a connection between it, the client, and the robot server. (Window-based applications automatically include the facility for users to make and break all the various client/server connections.)

3.1.1 Connecting a PAI Client with a Robot or the Simulator

int paiRobotStartup(int channel, char *name);

The PAI function **paiRobotStartUp** establishes a client/server connection to the physical robot through one of your computer's serial ports (typically COM1 or COM2), over a TCP/IP-based network, including the Internet, or through a "local" channel to a simulator or servers running on the same machine as the client (Table 3-1). Of course, the robot or simulator must be on and ready to make a connection with the client over the selected channel.

Table 3-1. Port types (channels) and Saphira-defined names for client/server connections

Channel	*sfLOCALPORT*	connect to simulator on the host machine
	sfTTYPORT	connect to robot on a tty port
	sfTCPPORT	connect to robot on over TCP/IP network
Port name	*sfCOMLOCAL*	local pipe or mailslot name (simulator or local server)
	sfCOM1	tty port 1 (/dev/ttya or /dev/ttyS0 for Unix/Linux; COM1 for MSW)
	sfCOM2	tty port 2 (/dev/ttyb or /dev/ttyS1 for Unix/Linux, COM2 for MSW)
	SERVER_NAME	hostname/IP address of server (not for Pioneer)

Once the connection is made, the paiRobotStartup function also intitiates other low-level processes that receive data from and send commands to the robot server or simulator. It also starts the robot's motor controllers, sonar pinging cycle, and other routine functions.

The port names sfCOM1 and sfCOM2 should represent the two lowest-numbered serial ports on the host computer. For example, on a FreeBSD system, sfCOM1 and sfCOM2 typcially represent */dev/ttyd0* and */dev/ttyd1*, respectively. If these variable names don't work, you can pass a string with the name of any desired port, as shown in the second and third examples above.

3.1.2 Connection Examples

```
paiRobotStartup (sfTTYPORT, sfCOM1); /* Connect through serial port COM1 */
paiRobotStartup (sfTTYPORT, "/dev/ttyS1");
paiRobotStartup (sfTCPPORT, "pioneer1.activmedia.com");
paiRobotStartup (sfLOCALPORT, sfCOMLOCAL); /* Connect with the Simulator */
```

3.1.3 Disconnecting from the Robot Server

Severing the client/server connection happens with a single PAI command, too:

void paiRobotShutdown (void);

3.1.4 Connect/Disconnect Example

```
/* This simple PAI program establishes a connection with, then right away disconnects
    from the simulated Pioneer robot. */

#include "pai.h"
void main (argv,argc)
{
        if (!sfConnectToRobot(sfLOCALPORT, sfCOMLOCAL))  /* Pioneer Simulator*/
```

```
    {
        printf("Couldn't open simulator!\n");
        exit(0);
    }

        paiRobotShutdown (void);    /* Shut it right down. */
}
```

3.2 Multiprocessing

PAI implements Saphira's multiprocessing capabilities of "lightweight" processes. By "lightweight" we mean that they each run without interruption, and must therefore execute very quickly to avoid a general system slowdown.

Saphira's cycle time is 100 milliseconds. Hence, every process in the system may run once every tenth of a second. Some simple math involving specifics of your host system can give you a rough estimate of how many processes of what complexity can run before they exceed the cycle period. In general, well-designed processes for behavior-based control should execute more than quickly enough to avoid problems; one must simply be careful to avoid any pauses (paiPause, sleep, wait, etc.), lengthy loops, large memory allocations, costly function calls, and other pitfalls. The option is always available to use separate "heavyweight" processes that communicate with your PAI code if your operating system provides this ability.

One must also be aware of the constraints of the robot as a physical system. Many processes might try to control, for example, one motor at the same time. This can lead to unpredictable and/or undesirable results. Many multiprocessing systems have intermediate processes that somehow integrate motor-control requests from other processes into the final commands that are sent to the motors.

3.2.1 Process Type

Processes run concurrently with Saphira/PAI main function. In WINDOW mode, they can be monitored through the Processes window, accessed from the main Function menu.

type process

The process type is generally used as a pointer. Pointers to processes let you integrate more general process-manipulation functions. We expect that with the simple nature of PAI multitasking it will be rare for users to do much direct process manipulation with such pointers.

3.2.2 Making Processes

macro paiMAKE_PROCESS(name, func *function, int delay)

The **paiMAKE_PROCESS** is a macro that properly packages a function to run as a PAI process. It provides handling of signals that cause suspension and activation of the process, and some very rudimentary scheduling.

The result of this macro is the global definition of name as a pointer to a process (**process*) which will call function *func* at the rate specified by *delay*. The argument *func* must be a function of no arguments and no return value, and which executes within the time constraints discussed above.

The argument *delay* is the number of 100ms cycles that should pass between calls to the function. That is, a delay of 0 means that the process will execute a full 10 times a second; delay=1 and the process will execute every other cycle, or five times per second; and so on.

The string argument *name* must be a unique identifier for the process.

A paiMAKE_PROCESS macro for each of your PAI processes must appear *only* at the top level of your C source PAI code, where function and global-variable definitions normally reside. See the example in Section 2.4 below for clarification, as needed.

3.2.3 Initializing Processes

void paiInitProcess(process *name, char *tag, initial_state)

const paiACTIVE
const paiSUSPENDED

The function **paiInitProcess** registers the *name* process with the system's multiprocessing scheduler, so that it will be executed. Use the unique string *tag* to later identify the process. It's also the one that appears in the Processes window of the GUI client.

Set the *initial_state* parameter to either **paiACTIVE** or **paiSUSPENDED**, depending on whether the process should run as soon as it is registered or remain in a suspended state until it gets activated externally, respectively.

3.2.4 Managing Processes

void paiActivateProcess(process *proc)

The function **paiActivateProcess** causes a process to become active. If the process already is active, the call has no effect. Too-frequent activation of a process can override the delay specified in **paiMAKE_PROCESS**.

void paiSuspendProcess(process *proc)

Not surprisingly, **paiSuspendProcess** functions to suspend a process. While suspended, a process won't get executed by the Saphira multiprocessing scheduler until some other process or your main program activates it with **paiActivateProcess**.

3.2.5 Multiprocessing Example

Okay, here's an example segment in which the main PAI program does some suspending and activating of processes:

```
/****************** *********/
/*******  Follow Color *******/
/****************************/

/* Program for following color blobs from channel a, using */
/* two simple processes. Requires Fast-Track Vision System. */

#include "pai.h"

int MinSonarDist (int low, int high) /* Returns closest sonar reading */
{                                    /* from among sonars low to high */
  int i;
  int closest = 5000;
  for (i=low; i <= high; i++)
    if (paiSonarRange(i) < closest)
      closest = paiSonarRange(i);
  return(closest);
}
void rotfunc (void)                /* This turns the robot towards   */
                                   /* the blob reported by channel a */
{
  paiRotateRobot (0.2 * DEG_TO_RAD * paiVisBlobX(CHANNEL_A));
}
void velfunc (void)                /* This controls velocity:      */
{                                  /* Fast if a blob is seen,      */
  if (MinSonarDist(1,5) < 400)     /* Slow if no blob is seen,     */
    paiSetRobotVelocity(0);        /* Stop if there is an obstacle */
  else
    if (paiVisBlobArea(CHANNEL_A) > 20)
      paiSetRobotVelocity(300);
    else
      paiSetRobotVelocity(100);
}

paiMAKE_PROCESS(TurnProc,rotfunc,1) /* set up a proc to run rotfunc  */
                                    /* needs at least a 1 cycle delay  */
                                    /* for result to be noticed */

paiMAKE_PROCESS(GoProc,velfunc,0)   /* set up a proc to run velfunc  */
                                    /* delay not needed because velocity */
                                        /* effects are "less drastic" */

void
main(int argc, char **argv)
{
  if (paiRobotStartup(sfTTYPORT, sfCOM2))  /* Start the robot */
    {
      paiInitProcess(TurnProc,"VEL",paiACTIVE); /* Start the 2 procs  */
      paiInitProcess(GoProc,"ROT",paiACTIVE);   /* they start off     */
    }                                           /* ACTIVE since we're */
  else                                          /* not putting in any  */
    printf("Can't connect!!\n");                /* code to call 'em.  */
}
```

3.3 Position and Motion

Pioneer operates in a Cartesian coordinate system measured in millimeters. Beginning at the origin (x=0, y=0, th=0) set when you first make a connection with the robot (**paiRobotStartup**) or if you reset its coordinate position (**paiResetRobotPosition**), the value for *x* increases as the robot travels to the right of the origin, and decreases as the robot travels left, becoming negative as the robot crosses over the origin line. Similarly, the *y* position parameter increases as the robot travels up and away from the origin, and decreases as the robot travels back towards the origin line.

The robot's heading, *th*, measured in degrees, is 0 when the robot starts up, is reset, or when it's rotational heading comes full circle around to that original orientation.

Note that the coordinate system is robot-centric, relative to its startup or reset position and heading. Use the PAI translational and rotation functions (following section) to actually move the robot and change its orientation.

float paiRobotX(void);

float paiRobotY(void);

The paiRobotX and paiRobotY functions return the x and y displacements in millimeters, respectively, or the robot relative to its **paiRobotStartup** or **paiResetRobotPosition** origin (x=0, y=0) position.

float paiRobotHeading(void);

The **paiRobotHeading** function returns the current heading of the robot in degrees. Again, because the coordinate system is robot-centric, the heading of 0 degrees is the relative rotational orientation of the Pioneer when you first started a connection with **paiRobotStartup**, or if and when you last reset it with **paiResetRobotPosition**.

void paiResetRobotPosition(void);

paiResetRobotPosition resets the robot's current position to (x=0, y=0, th=0). Accordingly, at that same moment, calls to **paiRobotX**, **paiRobotY**, and **paiRobotHeading** all would return the value 0.

3.3.1 Translation

PAI-controlled robot velocities are measured in millimeters per second. The maximum speeds attainable by a PAI controlled robot is platform dependent and limited. For instance, the Pioneer 1 has a maximum translational velocity of around 300 millimeters per second, depending on the surface and battery level, whereas its RoboCup and AT counterparts can achieve speeds over one meter per second. (Refer to your Pioneer's Saphira parameters file for robot-specific details.)

The robot's low-level systems work to maintain the velocity you set, but of course the velocity will vary due to current surface and terrain conditions, such as when initiating a climb or when first passing from a smooth to a carpeted floor.

Positive velocity values move the robot forward; negative values tell it to move backward.

void paiSetRobotVelocity(int vel);

Use **paiSetRobotVelocity** with the integer parameter *vel* to set a target velocity for the robot, in millimeters per second. The robot server will attempt to maintain this velocity throughout its travels.

Note that this command will cause the robot to begin moving—either forwards for positive values of *vel*, or backwards for negative values of *vel*—if it was initially stationary. Otherwise, the robot will speed up or slow down, or even stop and reverse its direction, depending on its prior velocity setpoint. So, also note that *vel* is absolute, not relative to a prior **paiSetRobotVelocity** setpoint.

void paiMoveRobot(int dist);

void paiSetMoveVelocity(int vel);

The function **paiMoveRobot** causes the Pioneer to move forward a positive distance *dist* (in millimeters) or backwards for negative *dist* and then stop. Use **paiSetMoveVelocity** to prescribe the **paiMoveRobot** translational rate *vel* in millimeters per second (initally set to the robot's maximum translational velocity).

Note that **paiMoveRobot** and its operating velocity you set with **paiSetMoveVelocity** override the effects of **paiSetRobotVelocity**, and the reverse. Accordingly, the robot complies with the most recent command and

"forgets" about any previous one. Conversely, translational commands work asynchronously with rotational ones, producing cummulative effects. Also see the section "Rotation" below.

float paiRobotTransVel(void);

So, just what is the robot's current translational velocity? Use **paiRobotTransVel** to find out. Again, the function's returned value is a snapshot of the robot's motion at the time and can vary depending on terrain and obstructions.

3.3.2 Rotation

PAI-controlled robot rotations are in degrees, usually performed at the robot's maximum rotation velocity (see the Saphira parameters file for your robot's specifications). Rotation commands of more than 360 degrees can have strange results. Positive rotations are counterclockwise; negative ones make the robot turn clockwise.

While one rotation command can override the effects of another one which has not completed, rotational controls may be performed simultaneously with robot translation commands without cancelling or interfering effects.

void paiRotateRobot(float degrees)

This function lets you rotate the robot a number of *degrees* relative to its current heading at the robot's maximum rotational speed. Careful: "Current heading" is the heading at the time you invoke this function. Robots take time to rotate, so if you issue a quick sequence of **paiRotateRobot** commands, you probably will not achieve their additive rotation (sum of individual *degrees*) but rather something less, even much less.

void paiSetRobotHeading(float degrees)

The **paiSetRobotHeading** function rotates the robot to an absolute heading *degrees* at the robot's maximum rotational speed. As with **paiRotateRobot**, this function may be performed together with translational commands with cummulative, not interfering or destructive, effects. But it does override the effects of previously issued, but not yet competed, rotation commands.

void paiSetRobotRotVel(float rotvel)

Use **paiSetRobotRotVel** to turn the robot at the set rate *rotvel* in degrees/second. The robot will continue to rotate at the prescribed rate until you issue another rotation (not translation) command or **paiStopRobot**.

float paiRobotRotVel(void);

Of course you occassionally will want to know if the robot is actually turning and at what rate. The **paiRobotRotVel** takes a snapshot of the robots current rotational velocity and returns that value in degrees per second. Like with translational velocities, be aware that rotational speeds can and will vary from moment to moment depending on terrain and other conditions.

3.3.3 Stopping the Robot

void paiStopRobot (void);

Of course, to stop the robot from moving or turning you may individually set the translational and rotational velocities of the robot to 0. The function **sfStopRobot** does just that in one easy step to painlessly and unconditionally overrides all translation and rotation commands and stop the robot.

3.4 Status

float paiRobotStatus(void);

Query for the general condition of you robot with paiRobotStatus. It returns one of the following PAI constants, which should be self-explanatory:

`paiSTATUSNOPOWER`
`paiSTATUSSTOPPED`
`paiSTATUSMOVING`

float paiRobotStall(void);

The paiRobotStall function returns a value greater than 0 if the robot server can and does detect a motor stalls. The Pioneer 1 and RoboCup Pioneer can and will detect motor stalls; the Pioneer AT cannot.

3.5 Sonars

PAI sonar pingers are numbered starting at 0. For example, in the Pioneer 1 and AT, they are numbered from 0 to 6, clockwise from the one nearest the LCD panel. Pioneer 1 and ATs also have the capacity for an eighth sonar (number 7), if the user attaches one.

int paiSonarRange(int Pinger);

The integer function **paiSonarRange** returns the distance, in millimeters, to the closest object detected by the specified *Pinger*. The minimum range is dependent on the sonar hardware, and is about 200 millimeters. The maximum range—the value returned if there is no object detected—is around 5 meters.

int paiSetSonarPolling(char *sequence);

You may set the sequence by which the robot server pings its sonars to give precendence to some, enable others, and ignore the rest. On startup, for example, Pioneer 1's and AT's seven sonars repetitively ping in order from 0 through 6; the eighth sonar is not initally included in the sequence. At the ping rate of 25 Hz, this means an individual sonar reading is taken once every 7*40 = 280 milliseconds. Hence, nearly a third of a second elapses before you get a fresh reading. Use **paiSetSonarPolling** to reorder the sequence to ping an individual sonar more often, or remove a sonar from pinging altogether.

The *sequence* parameter for **paiSetSonarPolling** must be a string of up to 12 characters. Each character in the string must be ASCII value 1+ sonar pinger number, typically represented in octal formal. The string sequence, of course, represents the ping sequence. If a particular sonar number is not included in the string, that sonar is not pinged and will not return fresh range values.

For example,

```
paiSetSonarPolling("\001\003\007\003");
```

causes the Pioneer to fire only its side and center pingers, with the center pinger being fired twice as often as the side pingers. Or turn off all the sonars to get rid of offending clicking (blinds Pioneer, though):

```
paiSetSonarPolling("");
```

Here's the polling command to include an eighth sonar in Pioneer's default sonar sequence:

```
paiSetSonarPolling("\001\002\003\004\005\006\007\010");
```

3.6 I/O

Pioneer Mobile Robots have eight digital input and eight digital output ports, an analog-to-digital (A/D) line, a timer, and an RC servo controller. The following PAI functions support the I/O capabilities of the Pioneer. Refer to the *Pioneer Operations Manual* for port locations and hardware connections.

Throughout, ports are number 0 through 7 and are byte-encoded corresponding with the least through the most significant bit. Accordingly, the byte-coded port specification `0x55` refers to all the odd ports.

3.6.1 Digital Ports

int paiDigIn(void);

int paiDigInBit(int bit);

The **paiDigIn** function returns the byte-encoded value of the eight digital input lines. Of course, a bit value of one (1) means the port is set (at Vcc), whereas the bit value of zero (0) means the port is at signal ground.

Use the convenience function **paiDigInBit** to query the state of a single digital input port. The function returns 0 or 1 according to the state of the specified input bit.

int paiDigOut(void);

int paiDigOutBit(int bit);

Similar to **paiDigIn** and **paiDigInBit**, **paiDigOut** returns the byte-encoded state of the eight digital output lines, and **paiDigInBit** returns a value of 0 or 1 depending on the state of the specified output bit. These functions do not alter the state of a I/O port.

void paiSetDigOut (int out_bits);

Use **paiSetDigOut** to specify the state of each and all the digital output ports, as encoded in the least significant byte of the *out_bits* integer. The effect is absolute, regardless of the output bit's previous state: If the corresponding bit is on and the paiSetDigOut bit is set, the output port will remain set. If the bit is on and the corresponding paiSetDigOut bit is off, the digital output port will be reset to off (0).

For example,

```
paiSetDigOut(0x33);
```

sets the odd digital ports 1, 3, 5, and 7 off (0) and turns on the even ones (0, 2, 4, and 6 set to 1 each).

Careful:

```
paiSetDigOut(33);
```

turns bits 0 and 5 on and turns all the others off.

void paiSetDigOutBit(int port, int state);

Change the state of a single digital output port, leaving all others alone, with **paiSetDigOutBit**. The function takes two arguments. The first, *port*, specifies the digital port number (0 through 7). The second parameter, *state*, specifies whether that port should be turned on (1) or off (0).

For example,

```
paiSetDigOutBit(2,1);
```

turns on digital I/O port number 1.

int paiSetDigOutMask(int bits_on, int bits_off);

Use **paiSetDigOutMask** to change the states of more than one, but less than all, of the digital output ports, leaving others unchanged. (Remember, paiSetDigOut sets the state for each and all the ports; sfSetDigOutBit sets the state for a single bit at a time.)

The two arguments for **paiSetDigOutMask** encode which bits to turn on (*bits_on*) and which bits to turn off (*bits_off*). All other bits, not included in either *bits_on* or *bits_off*, remain unchanged. For example,

```
paiSetDigOutMask(5,16);
```

turns on the digital output ports 0 and 2, and turns off port 4, without affecting ports 1, 3, 5, 6, or 7. Alternatively,

```
paiSetDigOutMask(3,3);
```

turns the first two ports off. See why?

void paiKick (int ms);

Sets the digital output port 0 to high for *ms* milliseconds. (Implemented for RWI's ill-fated RoboCup Kicker. Feel free to use otherwise.)

3.6.2 A/D Port

int paiAnalogIn(void);

Use **paiAnalogIn** to determine the voltage applied to the A/D port by some device. The returned value is 0 - 255, corresponding to an applied voltage of 0 - 5 VDC.

3.6.3 Timers

int paiInputTimer(void);

void paiStartTimer (int pin);

Pioneers have a built-in microsecond timer. When you call **paiStartTimer**, a pulse is sent out on the specified output port *pin*. Use this trigger the event to be timed. The time (in microseconds) it takes after such a trigger for a pulse to be received on the IDT (input timer) pin is reported by paiInputTimer.

3.6.4 Servo Port

void paiSetPulseWidth (int ms);

Sets the waveform for position-servo output on the ODT pin.

3.7 Miscellaneous

void paiPause(int ms);

The **paiPause** function suspends a process for the *ms* specified number of milliseconds, much like the common system-based **sleep** or **wait** library functions. The difference is that **paiPaise** does not interfere with Saphira multitasking. According, when you need to wait for a callback or other timed event, use **paiPause** so that low-level packet communications between the Pioneer and the client can continue to proceed smoothly.

float paiBatteryVoltage(void);

Returns the current battery voltage of the Pioneer, in volts.

4. PAI Fast-Track Vision

This chapter briefly describes the interface to the Fast-Track Vision System provided by PAI. Refer to the User's Manual and extensive documentation of the vision system supplied by Newton Research Labs, Inc. for vision system descriptions and details.

4.1 FTVS Constants and Definitions

```
#define BLOB_MODE 0
#define LINE_MODE 1
#define BLOB_BB_MODE 2
#define CHANNEL_A 0
# define CHANNEL_B 1
# define CHANNEL_C 2
```

The various precompiler definitions describe the modes and channels of the Fast-Track Vision System (FTVS), which is Newton Labs' high-performance cognachrome vision system integrated with the Pioneer Mobile Robot. As may be obvious in the definition names, the FTVS has three video channels, each of which may operate in one of three different vision-sensor modes: color blob tracking, line following, or expanded blob tracking.

4.2 FTVS Modes

void paiSetVisChannelMode (int channel, int mode);

int paiVisChannelMode (int channel);

Use the PAI function **paiSetVisChannelMode** to set the vision operation *mode* for a particular *channel*. Use its companion **paiVisChannelMode** function with *channel* argument to find out which mode the specified channel is set.

4.3 Video Select

The FTVS has a single video output port (Video Out RCA socket) on the robot's backpanel through which you may view its operations. Select which of the three channels you want to send to that port with:

void paiDisplayVisChannel (int channel);

4.4 Persistents

The FTVS provides a way to preserve its various settings. The following PAI function sets the vision persistent specified by string *P* to *val*. Consult the user's manual for persistent variable names and possible values.

void paiSetVisPersistent (char *P, int val);

Use the following functions to preserve and recall, respectively, the FTVS' current setting in non-volatile memory onboard the robot

void paiSaveVisSettings (void);

void paiRestoreVisSettings (void);

4.5 Color Training

void paiTrainVisChannel (int channel);

The PAI function **paiTrainVisChannel** with argument *channel* tells the specified channel of the FTVS to recognize the color of the object in the center of the camera's visual field. Used for all modes.

void paiShrinkChannel (int channel);

void paiExpandChannel (int channel);

Expand or reduce the range of colors detected by the specified channel to more precisely train the vision system in various environments.

void paiTrainVisChannelMore (int channel);

Expands the training of the specified channel by filling in blobs that were detected by previous training.

4.6 Retrieving Vision Information

4.6.1 Blob Modes

int paiVisBlobArea (int channel);

int paiVisBlobX (int channel);

int paiVisBlobY (int channel);

The individual **paiVisBlob** functions respectively return the area and centroid coordinates of the largest color blob detected by the specified FTVS *channel* when in `BLOB` or `BLOB_BB` mode.

For the expanded blob mode (BLOB_BB), two additional values are accessible from PAI:

int paiVisBlobBBHeight (int channel);

int paiVisBlobBBWidth (int channel);

The **Height** version of **paiVisBlobBB** function returns the height of the bounding box for the largest blob detected by the FTVS in the selected *channel*. The **Width** version, of course, returns the width of the largest blob.

4.6.2 Line Mode

int paiVisNearSlice (int channel);

int paiVisNumSlices (int channel);

int paiVisLineX (int channel);

Return the appropriate LINE_MODE data.

5. Gripper & Experimenter's Module

The Pioneer Experimenter's module contains several expansion enhancements to the system, including multiplexed analog-to-digital and RC servo ports, a speaker, and software-controllable indicators and switches. The Experimenter's Module electronics also are used to power and control the Pioneer Gripper accessory.

PAI contains a number of convenience functions for controlling the various features of the Pioneer Experimenter's Module and Gripper. Please consult the *Pioneer Gripper & Experimenter's Module Manual* for hardware and system details.

5.1 Gripper Functions

5.1.1 Motion and States

The Pioneer Server Operating System—Pioneer's onboard software—automatically controls the Gripper. You simply send it state commands and it does the work.

int paiGripperState (void);

void paiSetGripperState (int state);

```
#define paiGRIPPEROFF 0
#define paiGRIPPERUP 1
#define paiGRIPPERMOVING 2
#define paiGRIPPERMIDDLE 4
#define paiGRIPPEROPEN 5
```

The PAI GripperState functions let you set a desired *state* for the Gripper and monitor its activities. Acceptable states are: **paiGRIPPERUP**, in which the paddles are closed, possibly onto an object, and the Gripper is in its highest position; **paiGRIPPERMIDDLE**, in which the paddles are closed (object in hand, possibly) and raised about five centimeters off the floor; and **paiGRIPPEROPEN**, in which the Gripper paddles are in their lowest position and fully open.

Besides returning one of the states just mentioned, the **paiGripperState** query function may alternatively return **paiGRIPPERMOVING**, to indicate that the gripper is in transition between states, or **paiGRIPPEROFF**, to indicate that no Gripper is attached or otherwise has been unable to attain the state most recently set by **paiSetGripperState**.

5.1.2 Paddle Breakbeams

The Gripper's paddles contain front and rear breakbeams which let you detect if and when an object gets between them.

int paiGripperFrontBreakbeam (void);

int paiGripperRearBreakbeam (void);

```
#define paiBEAMCLEAR 0
#define paiBEAMOBSTRUCTED 1
```

The two PAI breakbeam functions query the state of the paddle breakbeams and return 0, if nothing is between, or 1 if there is an object obstructing the beam.

5.1.3 Bump Sensors

The tips of the paddles also contain bump sensors for contact sensing.

int paiGripperContact (void);

```
#define paiNOCONTACT 0
#define paiCONTACT 1
```

The paiGripperContact function queries the state of the bump sensors at the tips of the paddles, returning 0 if no contact, and 1 if either or both of the contacts are triggered.

5.2 Experimenter's Module I/O

The Experimenter's Module (included also with the Gripper) contains three LEDs (Gripper adds two more), two switches, and a speaker. All are under your software control through PAI.

5.2.1 LEDs

The Experimenter's Module has a green LED on each side of its face and an amber LED on the left side with the switches. The Gripper has a green LED at the tip of each paddle that parallel the state of their counterparts on the Module face.

void paiSetXModLED (int light, int state);

```
#define paiOFF 0
#define paiNONE 0
#define paiON 1
#define paiAMBERLED 4
#define paiRIGHTLED 8
#define LEFTLED 16
```

Use the **paiSetXModLED** function to set the state of one of the Expansion Module and associated Gripper LEDs to on or off.

void paiSetXModLEDs (int lights_on, lights_off);

The plural version of the PAI LED controller function lets you set the state of more than one LED at a time. The arguments *lights_on* and *lights_off* are masks that you create with bit-ANDed combinations of **paiAMBERLED**, **paiLEFTLED**, **paiRIGHTLED**, and **paiNONE**.

For example:

```
paiSetXModLEDs(paiAMBERLED & paiLEFTLED, paiNONE);
```

turns on the left and amber LEDs.

5.2.2 Switch and Button

On the left side of the Pioneer Experimenter's Module (included with the Gripper, too) there are a 2-position slide switch and a momentary-contact pushbutton.

int paiXModButton (void);

```
#define paiBUTTONPRESSED 0
#define paiBUTTONRELEASED 1
```

int paiXModSwitch (void);

```
#define paiSWITCHDOWN 0
#define paiSWITCHUP 1
```

The PAI XMod button and switch functions simply return the state of the respective user input devices.

5.3 Speaker

void paiSing (char *str);

Not much is known about the sounds that may emanate from your Experimenter's module, other than the string argument *str* may be up to 40 bytes long and is comprised of tone and duration pairs.

6. PAI Function Summary

Client-Server Connection	**Section**
int paiRobotStartup(int channel, char *name);	
void paiRobotShutdown (void);	
Processes	
macro paiMAKE_PROCESS(name, func *function, int delay)	
void paiInitProcess(process *name, char *tag, initial_state)	
void paiActivateProcess(sfprocess *proc);	
void paiSuspendProcess(sfprocess *proc);	
Position	
void paiResetRobotPosition (void);	
float paiRobotX(void);	
float paiRobotY(void);	
float paiRobotHeading(void);	
Translation	
float paiRobotTransVel(void);	
int paiRobotStall(void);	
int paiRobotStatus(void);	
void paiSetRobotVelocity(int vel);	
void paiSetMoveVelocity(int vel);	
void paiMoveRobot(int dist);	
void paiStopRobot(void);	
Rotation	
float paiRobotRotVel(void);	
void paiSetRobotRotVel(float vel);	
void paiRotateRobot(float degrees);	
void paiSetRobotHeading(float degrees);	
Sonars	
int paiSonarRange(int Pinger);	
void paiSetSonarPolling (char *sequence);	
I/O	
void paiUserIOCommand (int mask, int vals);	
int paiDigIn(void);	
int paiDigOut(void);	
void paiKick (int ms);	
int paiAnalogIn(void);	
int paiInputTimer(void);	
int paiDigInBit(int bit);	
int paiDigOutBit(int bit);	
void paiSetDigOut (int out_bits);	
void paiSetDigOutBit(int bit, int val);	
void paiSetDigOutMask(int on_mask, int off_mask);	
void paiStartTimer (int pin);	
void paiSetPulseWidth (int ms);	
Miscellaneous	
void paiPause(int ms);	
float paiBatteryVoltage(void);	
Fast-Track Vision	
void paiSetVisChannelMode (int channel, int mode);	
int paiVisChannelMode (int channel);	
void paiDisplayVisChannel (int channel);	

void paiSetVisPersistent (char *P, int val);	
void paiSaveVisSettings (void);	
void paiRestoreVisSettings (void);	
void paiTrainVisChannel (int channel);	
void paiTrainVisChannelMore (int channel);	
void paiShrinkChannel (int channel);	
void paiExpandChannel (int channel);	
int paiVisBlobArea (int channel);	
int paiVisBlobX (int channel);	
int paiVisBlobY (int channel);	
int paiVisBlobBBHeight (int channel);	
int paiVisBlobBBWidth (int channel);	
int paiVisNearSlice (int channel);	
int paiVisNumSlices (int channel);	
int paiVisLineX (int channel);	
Gripper	
int paiGripperState (void);	
void paiSetGripperState (int state);	
int paiGripperFrontBreakbeam (void);	
int paiGripperRearBreakbeam (void);	
int paiGripperContact (void);	
Experimenter's Module	
void paiSetXModLED (int light, int state);	
void paiSetXModLEDs (int lights_on, int lights_off);	
int paiXModButton (void);	
int paiXModSwitch (void);	
void paiSing (char *str);	

7. Index

R

S

T

V

W

X

Warranty & Liabilities

The developers and marketers of PAI and Saphira software shall bear no liabilities for operation and use with any robot or any accompanying software except that covered by the warranty and period. The developers and marketers shall not be held responsible for any injury to persons or property involving the PAI or Saphira software in any way. They shall bear no responsibilities or liabilities for any operation or application of the software, or for support of any of those activities. And under no circumstances will the developers, marketers, or manufacturers of PAI and Saphira take responsibility for or support any special or custom modification to the software.